Some good C++ practices for using the **art** framework

Marc Paterno
Scientific Computing Division/Fermilab
Revision 3

Contents

Thanks

I'd like to expression my appreciation for valuable input (over many years) from Walter Brown, Chris Green, Chris Jones, Jim Kowalkowski, and Rob Kutschke, for constructive input. Many of the good suggestions are theirs.

All the mistakes are my own.

1 Scope and intent of this document

This document is intended for an audience that has

- some programming experience,
- at least beginning familiarity with C++, and
- at least beginning familiarity with the **art**[1] framework.

The intent of this document is to help the reader avoid some of the more common mistakes made by those with little experience in C++, or in use of the **art** framework, or both.

---

[1]The **art** framework's home page is https://cdcvs.fnal.gov/redmine/projects/art.

C++ is a large and complex language, and so this presentation could be extended almost without limit. I've hit topics of particular interest to me.

2   Coding issues

2.1   Use good names.

Good names are *crucial* to the clarity of code. This goes for *functions*, *classes*, and *variables*. Your code will be written (and re-written, or modified) a few times. It will be read many times. Make it easy to read!

If your variable, class, and function names are good enough, your code will need little commenting. Very well-written code carries *few* comments, not many comments. The need to write extensive comments is very often a sign of bad name choices.

Good names are not always long. A loop index should be called something short, like `i`, not `thisLoopIndex`.

Be wary of re-using loop variable names in the same function. While the compiler will not complain, it can cause confusion for readers. If you follow §2.2, you'll avoid this naturally.

2.2   Avoid the blob-o-code.

Giant functions are hard to understand, and so are hard to get right. 200-line functions (or 2000-line functions!) are not rare in some code bases. In dozens of reviews, *I have never seen a well-written, understandable, correct function that is hundreds of lines long.* Please do not take this as a challenge to make yours the first. Have mercy on your colleagues, and make functions short enough to be understandable. Function calls (with small objects as parameters, and large objects passed by reference) are cheap.

Some guidelines:

- If your function contains more than one looping or `if` block, consider encapsulating each block as its own function.
- Considering declaring the small function `inline`.
- If your function contains blocks of code, set off by comments, consider making each commented section a function, with a good name that makes the comment superfluous.
- If you have *nested* control structures, consider making the inner one a function (well-named, of course).
- Consider using one of the algorithms of the Standard Library (from the Standard header `<algorithm>`, rather than writing a loop.

N.B.: in C++11, the ease-of-use of the Standard algorithms is dramatically improved over C++03.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm> // for std::copy and std::for_each
```

```
4   std::vector<Thing> v( ... ); // initialized somehow

6   // C++ 2003: explicit loop
7   for (std::vector<Thing>::const_iterator i = v.begin(), e = v.end();
8        i != e; ++i)
9     { std::cout << *i << "\n"; }

11  // C++ 2003: standard algorithm (std::copy).
12  // Must  #include <iterator> for std::ostream_iterator
13  std::copy(v.begin(), v.end(),
14            std::ostream_iterator<Thing>(cout, "\n"));

16  // C++ 2011
17  std::for_each(v.begin(), v.end(),
18                [](Thing const& x) {std::cout << x << "\n"; });

20  // C++ 2011, using algorithm from cetlib/container_algorithms.h
21  cet::for_all(v, [](Thing const& x) {std::cout << x << "\n";});
```

Listing 2.1: The joy of C++11.

2.3   Avoid bare use of `new`.

The expression `new X` allocates memory for, and then constructs (in that memory) an object of type `X`; the value of the expression is the address of the created object.

You should almost never have a call to `new` in your code. The few calls to `new` in your code should be in initialization of some smart pointer. Bare calls to `new` are the most common cause of memory leaks, because one must be careful to have a matched `delete` for *all possible code paths*, including those resulting from exception throws.

```
1   int* ip1 = new int(3);                    // bad:
2   std::shared_ptr<int>  ip2(new int(3)); // good
3   std::unique_ptr<int>  ip3(new int(3)); // good
4   std::auto_ptr<int>    ip4(new int(3)); // ok, but deprecated;
5                                          // prefer unique_ptr
```

Listing 2.2: Good and bad uses of `new`

Allocate every resource in a single code statement which initializes a manager object (*e.g.*, a smart pointer) to manage the resource.

Don't use bare pointers as data members! If you follow this rule, your class destructors will be empty—so the compiler-generated destructor will be correct, and you don't even have to write one.

2.4 Use RAII.

RAII stands for *resource allocation is initialization.* This is a generalization of §2.3. Memory is not the only resource; file handles, database handles, or anything that can get created and destroyed, is a *resource.* Allocation and deallocation of resources (*e.g.,* opening and closing of files) should be managed through the lifetime of *objects.* The object lifetime rules are enforced by the compiler, and use of RAII ensures no leaks of resources—even if exceptions are thrown. You do not need to write `try/catch` blocks.

```
1  #include <cerrno>
2  #include <cstdio>
3  #include <system_error>

5  struct FileMgr
6  {
7    std::FILE* const fp;
8    FileMgr(const char* name, const char* mode ) :
9    fp(std::fopen(name, mode)) {
10     if (!fp)
11       throw std::system_error(errno, std::system_category(), name);
12   }
13   FileMgr(FileMgr const&) = delete;
14   ~FileMgr() noexcept { std::fclose(fp); }
15 };

17 void some_function_that_might_throw(); // implementation not shown

19 void example() {
20   FileMgr f("myfile.txt", "w");
21   some_function_that_might_throw();
22   fprintf(f.fp, "Some silly text.\n");
23 } // fclose is called upon exit
```

Listing 2.3: A simple resource manager.

2.5 Be `const`-correct.

Use the compiler to help catch errors. `const` is one of the simplest ways to do this.

- If a "variable" should not change its value after initializtion, declare it to be `const`. Any mistaken attempt to change it will then cause a compilation error.
- If a member function does not change the state of the object on which it is called, declare it to be `const`.
- If an argument of a function is not to be modified by that function, declare it to be `const` (this is usually a reference-to-`const`: `const&`).

N.B.: This becomes even more important when programming for a multithreaded world.

2.6   Use standard containers. Prefer `std::vector`.

In C++, use of a C-style array is rarely the correct choice. Use standard containers: `std::vector`, `std::list`, `std::dequeue`, `std::set` and `std::map`. C++11 introduces a few more standard containers.

`std::vector` should be your first choice for a container, unless you have a clear, specific reason to choose something else.

`std::vector` guarantees that its objects are stored in contiguous memory locations.

Use `std::array` (or `boost::array`) if you require a container that has its size fixed at compilation-time.

Learn the interface of `std::vector`, especially the various constructors and the use of `reserve` and `resize`.

If you think you need `std::dequeue`, think again. It is a very special-purpose container; it is almost never the right choice.

I did not include `std::multiset` and `std::multimap` in the list of containers, because I almost never prefer them.

2.7   Always initialize objects. Initialize variables at the point of declaration.

This goes for both *stack objects* and *member objects* (data members). Leaving an undefined object often leads to "undefined behavior", which most often means eventual memory corruption, crashing, or both.

```cpp
double x(0.0);
std::vector<double> v(3, 1.1); // three elements, each is 1.1
// Here's how to initialize for-loop variables...
for (std::size_t i = 0, sz = v.size(); i!=sz; ++i) { ... }
```
Listing 2.4: Initialize variables when declared.

Following these rules, the bodies of constructors of classes you write will usually be *empty*. All the initialization should be done in the *initializer-list*.

```cpp
struct FileSentry {
  std::FILE* const fp;     // our data member
  FileSentry(const char* name, const char* mode ) :
     fp(std::fopen(name, mode)) // initializer-list (only 1 member)
  { ... }                       // implementation as shown earlier
  ...                           // remaining implementation as before
}
```
Listing 2.5: Using an initializer-list.

2.8   Use caution with problematic libraries.

Sometimes you have to use libraries that do not follow good C++ practice, such as avoidance of bare pointers. When presented with such libraries, a few defensive measures are in order:

- Be sure to understand each function you have to use. Does passing a pointer to a function pass ownership of the object pointed to, or does it not?
- Sometimes the answer is "it depends on the circumstances". In such a case, try to encapsulate the use of the dangerous resource.
- Sometimes, use of sentry objects (see §2.4) can avoid the problems inherent in a poorly designed interface.

2.9   Compiler-generated code will not contain errors.

Use compiler-generated copy constructors, copy assignment operators, and destructors whenever they do the right thing. Write your classes so they always do the right thing.

This becomes still more important with C++11, where the compiler, under the right conditions, may also supply a move constructor and a move assignment operator.

3   Design issues

3.1   Take a moment for design.

Each class, and each function, should have a clear purpose. Take a moment to think "*What* does this class (or function) do?". Don't think first of what a class *contains*, or how a function is *implemented*.

Some good rules of thumb:

1. For almost all classes, you should be able to express the essence of the class in one or two sentences, which do not make mention of implementation.
2. For almost all functions, you should be able to express the result of the function, without making mention of the implementation.

Example: `std::cos(double x)` calculates the cosine of the angle x, expressed in radians. Note there is no mention of lookup-tables, or calling of assembly language routines.

Since you have thought of this one- or two-sentence description of your class or function, put it at the top of the header as documentation for the class. This is probably the best concise documentation that can be provided for each class or function.

3.2   Plan for change.

Code is continually revised, updated, extended, and reused. Some up-front preparation for this makes future modifications easier.

For example, if you're introducing a second way to do something, plan for more in the future—unless other ways are logically impossible.

```
1  bool alg_b = ps.get<bool>("do_alg_b");
2  if (do_alg_b) { alg_b(); }
3  else          { alg_a(); }
```

<div align="center">Listing 3.1: Lack of planning. What if a third option is needed?</div>

```
1  using art::Exception; // to make lines shorter here
2  using art::errors::Configuration;
3  std::string alg = ps.get<std::string>("alg");
4  if (alg == "alg_a")      { alg_a(); }
5  else if (alg == "alg_b") { alg_b(); }
6  else { throw Exception(Configuration, "unknown alg")
7                   << alg); }
```

<div align="center">Listing 3.2: Good planning. New options would not break existing code.</div>

3.3 Don't over-generalize.

The previous point (§3.2) could have been labeled "Don't under-generalize". This point says to avoid the other extreme, *over-generalization*.

- Don't introduce infrastructure for multiple options when only one option exists.
- Don't introduce a base class when you will have only one derived class.
- Don't write a class or function template that will be instantiated for one specific type.

Abstractions always cost *some* mental overhead; introduce them when they are useful, but *only* where they are useful. Don't introduce them "just in case things change later".

3.4 Don't use inheritance to implement aggregation.

Use inheritance to introduce a type that can be *re-used*, by being implemented in several different ways. Example: in the framework, EDFilter defines an interface, and your filter classes inherit from it. EDFilter is used by the framework so that any class you derive from it can be used in any place that an EDFilter can be used.

If you want a class to contain an object of another type, add a data member of that type. Do not use inheritance.

Inheritance induces stronger *coupling* between classes, and makes it harder to change one without affecting the other. Why is this? The derived class has the *whole interface* of the class from which it inherits. A container class does not have the whole interface of any class it contains; it can choose what part, if any, of that interface to support.

An airplane may *contain* a pilot, but an airplane *is not substitutable for* a pilot.

4   art-specific issues

4.1   Use the module interface as designed.

All modules have a similar interface, which reflects the "lifecycle" of the event-processing loop.

- In the *constructor*, initialize all the module state that you can.
- In `beginJob`, initialize whatever could not be initialized at construction time. It is certainly safe to invoke *services* in `beginJob`. In `endJob`, clean up after anything that was initialized in `beginJob`.
- In `beginRun`, initialize whatever requires information available at the start of a new run, *e.g.*, the run number. Histograms that summarize the data for a run can be initialized here. At `endRun`, clean up after anything that was initialized at `beginRun`, *e.g.*, fitting or saving histograms. Clean up run-related things here, *not* at the next call to `beginRun`.
- `beginSubRun` and `endSubRun` are similar to `beginRun` and `endRun` (but for subruns, of course).
- In the *destructor*, clean up whatever was initialized in the constructor. N.B.: if you are following the suggestions above, your destructors are mostly empty. Then you don't have to write one, because the compiler-generated destructor will be correct.

4.2   Use the preferred form for `produce`.[2]

The interesting part of your module is the part that does the physics work, not the part that interacts with the framework. The recommended form for `produce` is:

```
1  void ThingFinder::produce(art::Event& ev) {
2    art::Handle<InputTypeOne> h1;
3    art::Handle<InputTypeTwo> h2;
4    ev.getByLabel("alg_a", h1);
5    ev.getByLabel("alg_b", h2);
6    std::auto_ptr<OutputType> prod(new OutputType());
7    // In this example, the first two arguments are inputs, the last
8    // is an output.
9    thingAlgorithm(*h1, *h2, *prod);
10   ev.put(prod);
11 }
12 // The signature of the putative function "thingAlgorithm" is:
13 // void
14 // thingAlgorithm(InputTypeOne const& a, // input, const
15 //                InputTypeTwo const& b, // input, const
16 //                OutputType& out);      // note: out is non-const
```

---

[2]and `filter`

```
17   // If a member function, it should probably be declared const.
```

Listing 4.1: Recommended style for writing `produce`.

Some results of this design are:

- If an exception is thrown anywhere, no memory gets leaked. The code is *exception safe*.

- If the required inputs are not found, we never even create the data product.

- The `ThingFinder` module handles all the framework-related tasks (interaction with the `Event`, `Handle`s, *etc.*

- The function `thingAlgorithm` knows about physics-related things, but isn't cluttered with framework-related things.

- There is a greater chance of re-using `thingAlgorithm` in another module type.

- It might be useful to make `thingAlgorithm` a member function in an algorithm `class` that can be used elsewhere; you'd then put `thingAlgorithm` in some library that can be shared between multiple modules.

As a bonus, this code will be relatively easy to make thread-safe, when doing so becomes important.

You may benefit from the command-line utility `artmod`.

```
1   ARTMOD(1)         User Contributed Perl Documentation         ARTMOD(1)

3   NAME
4          artmod: Generate clean module source for ART.
5   SYNOPSIS
6          artmod -h | --help | -?
7          artmod --help-types
8          artmod [optons] [--] module-type qualified-name
9          Options: --boilerplate|-b file |
10                   --entries|--entry|-e entry+ |
11                   --header-loc path |
12                   --split |
13                   --split-ext [lib-source-extension] |
14                   --verbose|-v
15          Options marked with + are repeatable and cumulative.
16   DESCRIPTION
17     artmod is a tool to produce an ART module source skeleton for an
18     analyzer, a filter or a producer. The user can specify which opt-
19     ional member functions are to be configured and whether to split
20     the source into three files or combine it into one. In addition,
21     the name of a file wherein boilerplate comments or code may be
22     found for insertion into the source may be specified.
```

Listing 4.2: Partial output from `artmod -h`.

5   Using C++11

5.1   Use the type-specifier `auto`.

Compare for clarity and brevity:

```
1  std::vector<std::string> names = ... // initialized somehow
2  for (std::vector<std::string>::const_iterator i = names.begin(),
3                                                 e = names.end();
4      i != e; ++i) {
5    do_something_with_a_name(*i);
6  }
```

<div align="center">Listing 5.1: Old-style for loop.</div>

and

```
1  std::vector<std::string> names = ... // initialize somehow
2  for (auto i = names.cbegin(), e = names.cend(); i != e; ++i) {
3    do_something_with_a_name(*i);
4  }
```

<div align="center">Listing 5.2: Using `auto`.</div>

Use `auto` to declare a variable whenever the compiler can use the initializer to deduce its
type.

```
1  auto x = std::cos(3.1);                           // auto -> double
2  auto const& y = ps.get<std::string>("n"); // auto -> std::string
3  auto z = std::qsort( ... );                       // illegal: returns void
```

5.2   Use the range-based `for` statement.

The new *range-based for statement* allows for simpler iteration over any "range". `std::string`,
Standard Library containers, and C-style arrays are all ranges. User-defined classes
which have right interface (supporting `begin` and `end`, and defining an *iterator*) are also
ranges, and can be used with the range-based for.

Compare the range-based for loop to the old-style for:

```
1  typedef std::map<std::string, std::string> dictionary;
2  dictionary definitions; // filling omitted
3
4  // C++2003 for loop
5  for (dictionary::const_iterator i = definitions.begin(),
6                                  e = definitions.end();
7      i != e;
8      ++i) {
9    std::cout << i->first << ": " << i->second << '\n';
10 }
```

```
12  // C++2011 range-based for loop
13  for ( auto const& d : definitions ) {
14    std::cout << d.first << ": " << d.second << '\n';
15  }
```

Listing 5.3: Range-based for loop.

6 Suggested reading

Two books I can recommend are:

1. For C++ coding advice: **C++ Coding Standards**, by Herb Sutter and Andrei Alexandrescu.
2. For object-oriented design advice: **Object-Oriented Design Heuristics**, by Arthur J. Riel. Some of the C++ suggestions in this book are dated, but the object-oriented design advice is excellent.

I can not yet recommend any good books on the use of C++11.